ICSC 2025 Qualification Round Solutions

Aditya Pal

September 2025

Contents

1. Problem A — Neural Network Components

For diagram of multilayer perceptron (MLP). Below is the explanation of each label:

a) $w_{21}^{(1)}$: Weight from **2nd input node** (Tip received) $\rightarrow$ **1st hidden node** in layer 1.

b) Σ: Represents the **weighted sum of inputs**, i.e.

$$z = \sum_j w_j x_j + b$$

c) f: **Activation function** - makes the output non-linear (e.g., **Sigmoid maps** values to 0–1 for probability, **ReLU** sets negatives to 0 and keeps positives unchanged).

d) Red Node: Represents an **input neuron**, i.e. the node where an **input parameter (feature) is fed into the network**.

e) Orange Node: Represents the **bias constant** – **a fixed value added in the summation** so that the neuron can still activate even if all inputs are zero, ensuring the network runs smoothly.

f) Green Node: Represents the **output neuron** – the **node that produces the final prediction** $\hat{y}$ after all computations in hidden layers.

g) Box A: Represents the **hidden layer**, where input features are transformed through weights, bias, and activation functions.

h) Box B: Represents the **output layer**, which combines signals from the hidden layer(s) to generate the final result/output.

i) $\hat{y}$: The **predicted value/output** of the network after processing all inputs.

2. Problem B — Cake Calculator

We are asked to calculate how many cakes can be made given an inventory of flour and sugar. The recipe requires:

$$\text{Flour per cake} = 100, \text{Sugar per cake} = 50$$

2.1 Approach

The limiting ingredient determines maximum cakes:

$$\text{cakes} = \min\left(\left\lfloor \frac{\text{flour}}{100} \right\rfloor, \left\lfloor \frac{\text{sugar}}{50} \right\rfloor\right)$$

Once the number of cakes is determined, the leftover flour and sugar are:

$$\text{flour\_left} = \text{flour} - 100 \times \text{cakes}$$

$$\text{sugar\_left} = \text{sugar} - 50 \times \text{cakes}$$

2.2 Python Implementation

Provided Code is Written by Me with reference to example codes given in website; (full .py file is uploaded as **problem-b.py**)

```python
# --- MY CODE HERE STARTS ----

# Ensure inputs are positive
if flour <= 0 or sugar <= 0:
   raise ValueError("Flour and sugar must be positive.")

# Constants for recipe
FLOUR_NEEDED = 100
SUGAR_NEEDED = 50

# Calculate maximum cakes based on limiting ingredient
cakes = min(flour // FLOUR_NEEDED, sugar // SUGAR_NEEDED)

# Calculate leftovers
flour_left = flour - (cakes * FLOUR_NEEDED)
sugar_left = sugar - (cakes * SUGAR_NEEDED)

# Return result
return [cakes, flour_left, sugar_left]

# --- MY CODE HERE ENDS ----
```

2.3 Example

If the input is:

$$\text{flour} = 450, \quad \text{sugar} = 120$$

Then:

$$\text{cakes} = 2, \quad \text{flour\_left} = 250, \quad \text{sugar\_left} = 20$$

3. Problem C — The School Messaging App

3.1 Context

According to the problem, we made a school messaging app that must work under the school's strict data limits. The data limits were small for each student, so **our challenge was to send the maximum number of characters in that limited bandwidth.**

3.2 Ques 1: Idea and Need of Variable Bit Lengths

Normally, text encodings give every character the same number of bits (e.g., 4 bits per letter), but this wastes space because some characters (like A and B) appear more often while others (like J and K) are rare.

To solve this, we applied **information entropy** and designed a **variable-length code**:

- Frequent characters get shorter codes (e.g., A = 000).
- Rare characters get longer codes (e.g., K = 1111).

3.3 Ques 2: Entropy Calculation

As given, entropy is calculated by:

$$H = -\sum_{i=1}^{n} p_i \log_2(p_i)$$

Where **H is entropy** and p_1 **is the probability of a character**.

The formula means we calculate $-p_i \log_2(p_i)$ for each character and sum them up to get the total entropy.

For each character:

$$A: \ -0.20 \cdot \log_2(0.20) = 0.4644$$

$$B: \ -0.15 \cdot \log_2(0.15) = 0.4105$$

$$C: \ -0.12 \cdot \log_2(0.12) = 0.3671$$

$$D: \ -0.10 \cdot \log_2(0.10) = 0.3322$$

$E: \ -0.08 \cdot \log_2(0.08) = 0.2915$

$F: \ -0.06 \cdot \log_2(0.06) = 0.2435$

$G: \ -0.05 \cdot \log_2(0.05) = 0.2161$

$H: \ -0.05 \cdot \log_2(0.05) = 0.2161$

$I: \ -0.04 \cdot \log_2(0.04) = 0.1858$

$J: \ -0.03 \cdot \log_2(0.03) = 0.1518$

$K: \ -0.02 \cdot \log_2(0.02) = 0.1129$

$L: \ -0.10 \cdot \log_2(0.10) = 0.3322$

Now, summing all contributions:

$H = 0.4644+0.4105+0.3671+0.3322+0.2915+0.2435+0.2161+0.2161+0.1858+0.1518+0.1129+0.3322$

$$H \approx 3.3240 \text{ bits/character.}$$

This means that the **theoretical minimum** average number of bits per character (for this distribution) is about 3.3240.

3.4 Ques 3: Average Length of Fano Code and Efficiency

Interpretation: Entropy vs. Fano Code

Entropy gives the **theoretical lower bound** on the average number of bits per character.
Since it is an average, entropy can have a **decimal value** (e.g., 3.324 bits/character).
In practice, code lengths must be integer bits, so we can't use 3.324 bits per character.

To approximate this, we construct a **variable-length code** such as the Fano code:

- Frequent characters are given shorter codes (e.g., 3 bits).
- Less frequent characters are given longer codes (e.g., 4 bits).

Thus, the Fano code is a **practical implementation** of entropy coding, and its average code length $\bar{L}$ is close to the entropy limit H.

Using the given Fano codes, the average code length is computed as:

$$\bar{L} = \sum_{i=1}^{n} p_i \cdot l_i$$

where p_i is the probability of character i and l_i is the code length (in bits) assigned to that character.

Calculation

$$\begin{aligned}\bar{L} &= (0.20 \cdot 3) + (0.15 \cdot 3) + (0.12 \cdot 3) + (0.10 \cdot 4) + (0.08 \cdot 4) \\ &\quad + (0.06 \cdot 4) + (0.05 \cdot 3) + (0.05 \cdot 4) + (0.04 \cdot 4) + (0.03 \cdot 4) \\ &\quad + (0.02 \cdot 4) + (0.10 \cdot 4) \\ &= 1.56 + 1.92 \\ &= 3.4800 \text{ bits/character.}\end{aligned}$$

Efficiency

$$\bar{L} - H = 3.4800 - 3.3240 = 0.1560 \text{ bits/character.}$$

$$\eta = \frac{H}{\bar{L}} = \frac{3.3240}{3.4800} \approx 0.9552 \quad (95.5\%).$$

Therefore, the Fano code achieves about 95.5% efficiency and significantly improves over fixed-length encoding (4 bits/character).

3.5 Conclusion

- Fixed-length coding needs 4 bits/character.
- Fano coding drops it to 3.48 bits/character.
- Efficiency is about 95 percent!

Our messaging app can send more characters within the data limit using entropy-based variable-length coding, making it super efficient!

4. Problem D — Word Search Puzzle Generator

4.1 Concept

We are asked to generate a word search puzzle of size 10×10 based on a list of input words without positional restrictions. The goal is to place the words horizontally, vertically, or diagonally in the grid, and fill empty cells with random letters. We generate a word search puzzle using **string manipulations**, iterations, and **two-dimensional lists** to build it. The steps are:

1. Create an empty 10 × 10 grid using 2D lists..
2. Place each word:
   - Pick a random direction (horizontal, vertical, or diagonal).
   - Choose a valid starting position so the word fits (Word can be placed in reverse order also).
   - Use loops to place the word's characters, breaking it down with string manipulation.
3. Fill blanks with random A–Z letters.
4. Print puzzle.

4.2 What Makes This Puzzle Different?

This isn't a "straight-line easy find." You need focus, like playing chess:

- Words can be reversed (e.g., FUN as NUF) using **Python's word[::-1] slicing**.
- It's not a piece of cake; it takes patience to spot them!
- Random placement makes every puzzle fresh — no memorizing here!

4.3 Python Implementation

Provided Code is Written by Me with reference to example codes given in website; (full .py file is uploaded as **problem-d.py**)

```python
# --- MY CODE HERE STARTS ----
    # Initializing a 10x10 grid with empty spaces to start building the puzzle
    grid = [[" " for _ in range(10)] for _ in range(10)]
    placements = []  # List to store placement details for each word
```

```
# Iterating through each word to place it in the grid
for word in words:
    word = word.upper()  # Converting word to uppercase for consistency
    if len(word) > 10:  # Skipping words longer than 10 characters
        continue

    placed = False  # Flag to track if word is successfully placed
    attempts = 0  # Counter for placement attempts
    max_attempts = 50  # Maximum number of attempts to place a word

    # Continuing to try placing the word until successful or max attempts
    ↪  reached
    while not placed and attempts < max_attempts:
        direction = random.choice(["horizontal", "vertical", "diagonal"])  #
        ↪  Randomly choosing placement direction
        reverse = random.choice([True, False])  # Deciding if word should be
        ↪  reversed
        word_to_place = word[::-1] if reverse else word  # Reversing word if
        ↪  chosen
        placement = {"word": word, "start_row": 0, "start_col": 0,
        ↪  "direction": direction, "reversed": reverse}  # Initializing
        ↪  placement dictionary

        if direction == "horizontal":
            # Randomly selecting starting position ensuring word fits
            ↪  horizontally
            row = random.randint(0, 9)
            col = random.randint(0, 10 - len(word))
            can_place = True  # Flag to check if position is valid
            # Checking if the position is valid (no conflicting letters)
            for i, char in enumerate(word_to_place):
                if grid[row][col + i] != " " and grid[row][col + i] != char:
                    can_place = False
                    break
            if can_place:
                # Placing the word in the grid
                for i, char in enumerate(word_to_place):
                    grid[row][col + i] = char
                placement["start_row"] = row
                placement["start_col"] = col
                placements.append(placement)  # Recording placement details
                placed = True

        elif direction == "vertical":
            # Randomly selecting starting position ensuring word fits
            ↪  vertically
            row = random.randint(0, 10 - len(word))
```

```
                col = random.randint(0, 9)
                can_place = True
                # Checking if the position is valid
                for i, char in enumerate(word_to_place):
                    if grid[row + i][col] != " " and grid[row + i][col] != char:
                        can_place = False
                        break
                if can_place:
                    # Placing the word in the grid
                    for i, char in enumerate(word_to_place):
                        grid[row + i][col] = char
                    placement["start_row"] = row
                    placement["start_col"] = col
                    placements.append(placement)
                    placed = True

            elif direction == "diagonal":
                # Randomly selecting starting position ensuring word fits
                ↪   diagonally
                row = random.randint(0, 10 - len(word))
                col = random.randint(0, 10 - len(word))
                can_place = True
                # Checking if the position is valid
                for i, char in enumerate(word_to_place):
                    if grid[row + i][col + i] != " " and grid[row + i][col + i]
                    ↪   != char:
                        can_place = False
                        break
                if can_place:
                    # Placing the word in the grid
                    for i, char in enumerate(word_to_place):
                        grid[row + i][col + i] = char
                    placement["start_row"] = row
                    placement["start_col"] = col
                    placements.append(placement)
                    placed = True
            attempts += 1  # Incrementing attempt counter

    # Filling any remaining empty spaces with random uppercase letters
    for row in range(10):
        for col in range(10):
            if grid[row][col] == " ":
                grid[row][col] = random.choice(string.ascii_uppercase)

    return (grid, placements)  # Returning the grid and placement details as a
    ↪   tuple
    # --- MY CODE HERE ENDS ----
```

Example Run

Suppose the user gives the words:

{LEARNING, SCIENCE, FUN}

One possible arrangement generated by the algorithm could look like this (example only, since the placement is random every time, Also I have used formattings to give it grid feel as demostration only, since skeleton code said not to Edit the Output part!):

```
+---+---+---+---+---+---+---+---+---+---+
| L | E | A | R | N | I | N | G | F | U |
+---+---+---+---+---+---+---+---+---+---+
| S | . | . | . | . | . | . | . | . | N |
+---+---+---+---+---+---+---+---+---+---+
| C | . | . | . | . | . | . | . | . | . |
+---+---+---+---+---+---+---+---+---+---+
| I | . | . | . | . | . | . | . | . | . |
+---+---+---+---+---+---+---+---+---+---+
| E | . | . | . | . | . | . | . | . | . |
+---+---+---+---+---+---+---+---+---+---+
| N | . | . | . | . | . | . | . | . | . |
+---+---+---+---+---+---+---+---+---+---+
| C | . | . | . | . | . | . | . | . | . |
+---+---+---+---+---+---+---+---+---+---+
| E | . | . | . | . | . | . | . | . | . |
+---+---+---+---+---+---+---+---+---+---+
| F | U | N | . | . | . | . | . | . | . |
+---+---+---+---+---+---+---+---+---+---+
| . | . | . | . | . | . | . | . | . | . |
+---+---+---+---+---+---+---+---+---+---+
```

Notice how:

- The word `LEARNING` was placed horizontally.
- The word `SCIENCE` was placed vertically.
- The word `FUN` was placed horizontally at the bottom.

The dots represent filler characters that will later be replaced with random letters to camouflage the words.

Conclusion

So, what did I really learn here? The main takeaways are:

- Using **2D lists** to represent a grid makes puzzle-building much easier.
- **Iterations** (nested loops) help place characters one-by-one into the right cells.
- **String manipulation** (like slicing and reversing) gives flexibility to place words in forward or reverse order.
- Randomization makes every puzzle fresh and challenging.

5. Problem E – Functional Completeness of NAND Logic Gate

Introduction

So our point here is to understand what logic gates are and why **NAND** is so special. Logic gates are basically circuit arrangements made using transistors that take advantage of semiconductor properties like voltage sensitivity. The inputs and outputs are always in **binary** (0 or 1).

The name itself says "logic" – they follow logical rules. And we have a few main ones:

- **AND = AND Gate (All inputs must be 1 to get 1)**
- **OR = OR Gate (At least one 1 is enough to get 1)**
- **NOT = Inverter (Just flips the value)**

Apart from these basics, we also have what are called **Universal Gates**:

- **NAND = NOT AND**
- **NOR = NOT OR**

Here comes the important term: **NAND is functionally complete**. That means with just NAND gates, we can construct *all possible logic gates and functions*. So in theory, if you only had NAND gates in the world, you could still design a computer! That is why NAND (and also NOR) are so important in electronics. Engineers prefer them in real life because they are cheaper to fabricate and flexible.

Truth Tables and Binary Behavior

Let's look at how the gates behave:

NOT Gate (Inverter):

A	NOT(A)
0	1
1	0

Example: If a bulb is ON (1), NOT turns it OFF (0). If OFF, NOT turns it ON.

AND Gate (All inputs must be 1):

A	B	$A \wedge B$
0	0	0
0	1	0
1	0	0
1	1	1

Example: A light turns ON only if both the switch and the sensor are ON.

OR Gate (At least one input 1):

A	B	$A \vee B$
0	0	0
0	1	1
1	0	1
1	1	1

Example: A room light can be turned ON by either switch A or switch B.

NAND Gate (NOT AND):

A	B	NAND(A, B)
0	0	1
0	1	1
1	0	1
1	1	0

Example: NAND is opposite of AND – it only gives 0 when both inputs are 1, otherwise always 1.

Making Other Gates from NAND (Proof of Functional Completeness)

Our point here is: **NAND is universal and functionally complete**. Let's prove it with clear examples and derivations.

- **NOT from NAND:** Connect both inputs of a NAND gate to the same signal.

  $$\textbf{NOT}(A) = \textbf{NAND}(A, A)$$

  Example: Consider $A = 1$:

  $$\text{NOT}(1) = \text{NAND}(1, 1)$$

  Since $\text{NAND}(1, 1) = 0$ (as NAND outputs 0 only when both inputs are 1), we get:

  $$\text{NOT}(1) = 0$$

  This **matches the NOT gate behavior** (inverting the input), hence proved. Thus, **NAND can function as an inverter**.

- **AND from NAND:** Apply NAND to the inputs, then **invert the result using the NOT trick**.

  $$A \wedge B = \textbf{NAND}(\textbf{NAND}(A, B), \textbf{NAND}(A, B))$$

  Example: Consider $A = 0$, $B = 1$:

  $$\text{AND}(0, 1) = \text{NAND}(\text{NAND}(0, 1), \text{NAND}(0, 1))$$

  First, $\text{NAND}(0, 1) = 1$ (since NAND outputs 1 unless both inputs are 1). Then, $\text{NAND}(1, 1) = 0$. So:

  $$\text{AND}(0, 1) = 0$$

  This aligns with AND behavior (output 0 when one of inputs is 0), hence proved.

- **OR from NAND: Using De Morgan's Law:** $A \vee B = \textbf{NOT}(\textbf{NOT}(A) \wedge \textbf{NOT}(B))$**.**

  Substituting the NOT operation with NAND:

  $$\text{NOT}(A) = \text{NAND}(A, A), \quad \text{NOT}(B) = \text{NAND}(B, B)$$

  Then,

  $$(\text{NOT}(A) \wedge \text{NOT}(B) = \text{NAND}(\text{NAND}(A, A), \text{NAND}(B, B))$$

Now **inverting this with another NAND gives**:

$$A \lor B = \text{NAND}(\text{NAND}(\text{NAND}(A,A),\text{NAND}(B,B)),\text{NAND}(\text{NAND}(A,A),\text{NAND}(B,B)))$$

However, a simpler form **using the NOT trick directly** is:

$$A \lor B = \text{NAND}(\text{NAND}(A,A),\text{NAND}(B,B))$$

Example: Consider $A = 0$, $B = 1$:

$$\text{OR}(0,1) = \text{NAND}(\text{NAND}(0,0),\text{NAND}(1,1))$$

$\text{NAND}(0,0) = 1$, $\text{NAND}(1,1) = 0$, then $\text{NAND}(1,0) = 1$. **This matches OR behavior (output 1 if at least one input is 1), hence proved.**

Awareness: Other Logic Gates

Apart from these, just to be aware:

- **NOR = NOT OR (also functionally complete)**
- **XOR = Exclusive OR (1 only when inputs are different)**
- **XNOR = Exclusive NOR (1 only when inputs are same)**

Our takeaway here is: once we know NAND is **functionally complete**, we are confident that we can build any of them using only NAND.

Conclusion

So in the end, the key point is:

> **NAND is functionally complete. That means it can generate NOT, AND, OR and even more complex ones like XOR. In short, NAND alone is enough to build the full universe of logic circuits.**

And honestly, that's the cool part — one single gate type, and you can design an entire computer!